Diseñando la fase final:

Para esta fase y diseño nos enfocamos en los siguientes requisitos:

- Implementar un algoritmo de generación de secuencias de llamadas y secuencia de retornos de procedimientos, i.e., traducir el Código de Tres Direcciones a código de MIPS que sea capaz de hacer saltos hacia y desde procedimentos (funciones) sin perder el estado general de la memoria, i.e., manejo del stack pointer.
- Implementar un algoritmo de asignación de registros o uso de pila, i.e., implementar la famosa función getReg() para asignar registros libres en los que puedan traducir su código intermedio a MIPS y realizar la asignación apropiada y adecuada para los tipos de registros que existen de MIPS, tales como los $t, $s, etc., así como la opción de utilizar el stack para guardar todas sus operaciones, i.e., manejo de la memoria y registros como tal.
- Generar código assembler en MIPS para su posterior ejecución por medio de una tercera herramienta, i.e., utilizar un simulador de MIPS para correr el código y validar que este se ejecute correctamente.

Eso sí, en nuestro caso hay algunos pasos iniciales que debemos mejorar de la fase anterior antes de seguir con la implementación de la fase de compilación.

Arreglos que son necesarios:

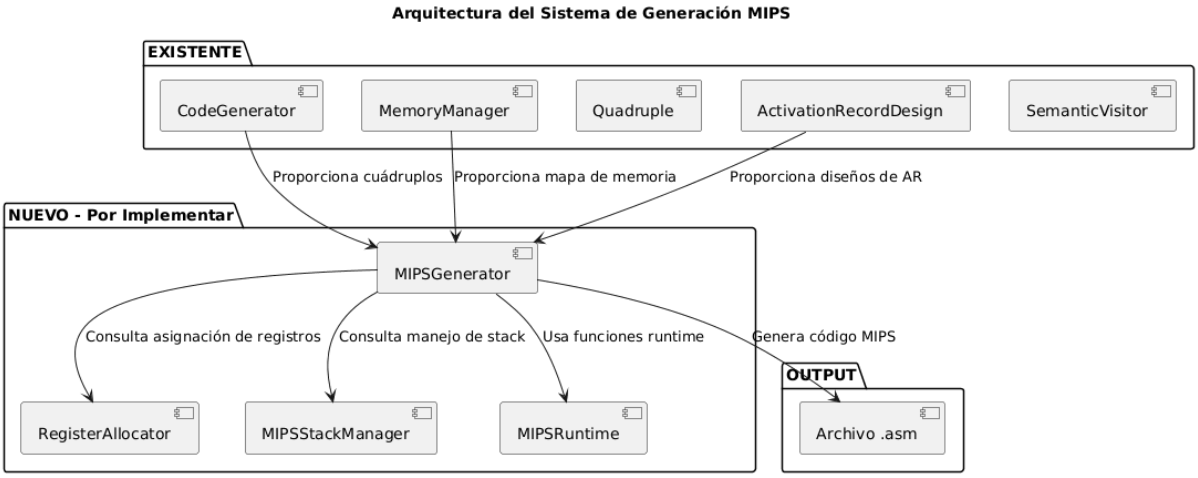
- Reducir si es posible la cantidad de cuádruplos que hemos generado y especialmente la cantidad de variables temporales. Aunque el código intermedio es funcional, es necesario que hagamos la mejora y optimización para poder usar mucho menos temporales.
- Además de esto es necesario estar completamente seguro del manejo correcto de funciones, clases y objetos con punteros y demás para poder generar un buen código que sea sencillo adaptar a mips.

Plan para generación de MIPS:

A grandes razgos lo que necesitamos hacer es:

- Traductor TAC->MIPS que convierta tus cuádruplos a instrucciones ejecutables
- Register Allocator que mapee temporales a registros físicos
- Stack Manager que maneje llamadas a funciones
- MIPS Assembler que genere el archivo final .asm

Para esto necesitaremos hacer nuevos archivos para traducción de TAC a MIPS, manejo de código y factores importantes como el stack por ejemplo. Las cosas necesarias para la siguiente fase se describen en el siguiente diagrama:



Código plant uml del diagrama:

```
@startuml

title Arquitectura del Sistema de Generación MIPS


package "EXISTENTE" {

  [CodeGenerator] as CG

  [MemoryManager] as MM

  [Quadruple] as QUAD

  [ActivationRecordDesign] as AR

  [SemanticVisitor] as SV

}


package "NUEVO - Por Implementar" {

  [MIPSGenerator] as MIPSG

  [RegisterAllocator] as RA

  [MIPSStackManager] as SM

  [MIPSRuntime] as RT

}
```

package "OUTPUT" {

[Archivo .asm] as ASM

}

CG --> MIPSG : Proporciona cuádruplos

MM --> MIPSG : Proporciona mapa de memoria

AR --> MIPSG : Proporciona diseños de AR

MIPSG --> RA : Consulta asignación de registros

MIPSG --> SM : Consulta manejo de stack

MIPSG --> RT : Usa funciones runtime

MIPSG --> ASM : Genera código MIPS

@enduml

1. El MIPSGenerator será el corazón de esta fase nueva, donde buscaremos poder hacer la traducción de los cuadruplos y generación de coódigo MIPS en general utilizando los componentes de la fase anterior también.
Una implementación base es la siguiente:

```python
class MIPSGenerator:
    def __init__(self, code_generator, symbol_table):
        self.cg = code_generator
        self.symbol_table = symbol_table
        self.register_allocator = RegisterAllocator()
        self.stack_manager = MIPSStackManager()
        self.runtime = MIPSRuntime()

    def generate_mips_code(self):
        """Genera código MIPS completo desde los cuádruplos"""
        # 1. Sección de datos
        data_section = self._generate_data_section()

        # 2. Sección de texto con funciones
        text_section = self._generate_text_section()

        # 3. Ensamblar archivo final
        return self._assemble_final_code(data_section, text_section)
```

2. Necesitamos también el register_allocator.py, esto servira para la asignación de rgistros y para poder implementar el algoritmo de getReg(). Además de ser muy útil de la asignación de registros:

```
class RegisterAllocator:
    def __init__(self):
        self.available_regs = {
            'temp': [f'$t{i}' for i in range(10)],
            'saved': [f'$s{i}' for i in range(8)],
            'arg': [f'$a{i}' for i in range(4)]
        }
        self.used_regs = {}
        self.spill_offset = 0

    def get_reg(self, temp_name, context='arithmetic'):
        """Implementa el algoritmo getReg() famoso"""
        # Lógica de asignación inteligente
        pass

    def free_reg(self, temp_name):
        """Libera un registro para reutilización"""
        pass
```

3. Crearemos un mips_stack_manager.py para el manejo del stack, que será útil para la secuencia de llamadas y retornos de funciones. Esta será una base para comenzar:

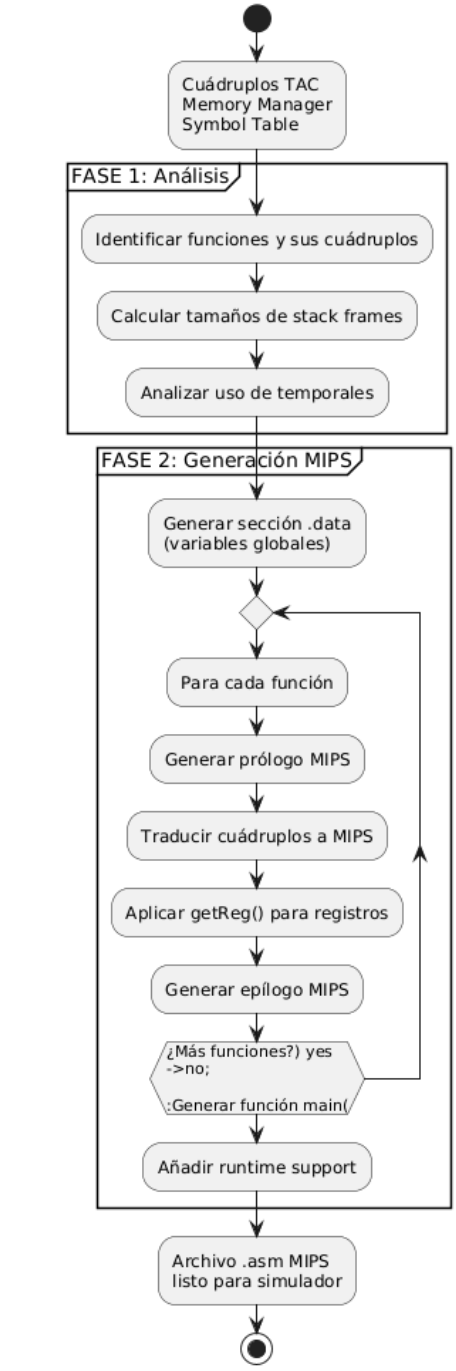
```
class MIPSStackManager:
    def generate_function_prologue(self, func_name, local_vars_count):
        """Genera secuencia de entrada a función"""
        pass

    def generate_function_epilogue(self, func_name):
        """Genera secuencia de salida de función"""
        pass

    def generate_call_sequence(self, func_name, arg_count):
        """Genera secuencia para llamar función"""
        pass
```

4. También es necesario un mips_runtime.py para el soporte de ejecución del código MIPS, ya que MIPS tiene funciones auxiliares como prints y demás.

```
class MIPSRuntime:
    def get_runtime_functions(self):
        """Devuelve código MIPS para funciones de runtime"""
        return [
            "print_int: li $v0, 1; syscall; jr $ra",
            "print_string: li $v0, 4; syscall; jr $ra",
            # ...
        ]
```

Viendo el diagrama de traducción completo sería de la siguiente manera:

Proceso de Traducción de Cuádruplos a MIPS



Para poder elaborar esto es necesario separar en etapas. La traducción de código TAC a MIPS no es tan sencillo como un reemplazo con varios ifs, esta traducción implica varias cálculos y manejo de estructuras para que se haga de manera apropiada.

Etapa1: Traduccion de operaciones elementales (CHUY)

Con esto nos referimos por ejemplo a la traducción de los cuádruplos aritméticos y de control básicos por ejemplo.

```
# En mips_generator.py
def translate_arithmetic_quad(self, quad):
    """Traduce cuádruplos aritméticos : (*, t74, 2, t75)"""
    arg1_reg = self.ra.get_reg(quad.arg1)
    arg2_reg = self.ra.get_reg(quad.arg2)
    result_reg = self.ra.get_reg(quad.result)

    mips_op = {'+': 'add', '-': 'sub', '*': 'mul', '/': 'div'}[quad.op]

    return [
        f"# {quad.comment}" if quad.comment else "",
        f"lw {arg1_reg}, {self.get_memory_address(quad.arg1)}",
        f"lw {arg2_reg}, {self.get_memory_address(quad.arg2)}",
        f"{mips_op} {result_reg}, {arg1_reg}, {arg2_reg}"
    ]

def translate_assignment_quad(self, quad):
    """Traduce cuádruplos de asignación como: (=, t78, None, 0x1030)"""
    value_reg = self.ra.get_reg(quad.arg1)
    target_addr = self.get_memory_address(quad.result)

    return [
        f"# {quad.comment}",
        f"lw {value_reg}, {self.get_memory_address(quad.arg1)}",
        f"sw {value_reg}, {target_addr}"
    ]
```

Etapa 2: Funciones y Stack management (JAPO)

Esto será para la secuencias de llamadas y retornos. MUY importante tener en cuenta que aquí también sería necesario poder manejar la recursividad. Loos snippets de código es más que todo para la definición inical de las clases y ejmplos básicos de las cosas por hacer.

```
# En mips_stack_manager.py
def generate_function_prologue(self, func_name, local_size, param_count):
    """Genera código para entrada a función"""
    return [
        f"# PROLOGUE - {func_name}",
        "addiu $sp, $sp, -8     # Reserve space for $ra and $fp",
        "sw $ra, 4($sp)         # Save return address",
        "sw $fp, 0($sp)         # Save frame pointer",
        "move $fp, $sp          # Set new frame pointer",
        f"addiu $sp, $sp, -{local_size} # Space for local variables"
    ]

def generate_call_sequence(self, func_name, args):
    """Genera código para llamar función"""
    instructions = ["# CALL SEQUENCE"]

    # Pasar argumentos en $a0-$a3 y stack
    for i, arg in enumerate(args[:4]):
        instructions.append(f"move $a{i}, {arg}")

    for i, arg in enumerate(args[4:], 4):
        instructions.append(f"sw {arg}, {i*4}($sp)")

    instructions.append(f"jal FUNC_{func_name}")
    return instructions
```

Etapa 3: Objetos y memoria (NELSON)

Esto tiene como prinvipal tarea el acceso a objetos y llamadas a los métodos

```
def translate_object_access(self, quad):
    """Traduce acceso a objetos como: 284: (+, t72, 8, t73)"""
    if quad.op == '+' and 'Address of' in str(quad.comment):
        # Es un cálculo de offset de objeto
        base_reg = self.ra.get_reg(quad.arg1)
        result_reg = self.ra.get_reg(quad.result)
        offset = quad.arg2  # ej: 8 para Estudiante.edad

        return [
            f"# {quad.comment}",
            f"addiu {result_reg}, {base_reg}, {offset}"
        ]

def translate_method_call(self, quad):
    """Traduce llamadas a métodos como: 306: (call, None, None,
FUNC_promedioNotas)"""
    # Tu código ya tiene push de argumentos antes del call
    # Solo necesitamos traducir call a jal
    return f"jal {quad.result}"
```

Final: Prueba de código generado

Necesitamos también una función que ensamble todos los campos necesarios de un programa MIPS. A grandes razgos sería algo como esto:

```
def generate_complete_program(self):
    sections = []

    # 1. Encabezado
    sections.append("# Generated by Compiscript Compiler")
    sections.append("")

    # 2. Sección data
    sections.append(".data")
    sections.extend(self.generate_global_variables())
    sections.append("")

    # 3. Sección text
    sections.append(".text")
    sections.append(".globl main")
    sections.extend(self.generate_all_functions())
```

```
    sections.extend(self.generate_runtime_support())
    return "\n".join(sections)
```

Al final de esto sería necesario probar el código mips que generamos con nuestro programa e IDE con un simulador. Tomando en cuenta la estructura del proyecto que hemos manejado y el resultado final esperado, idealmente usaríamos el simulador MARS para probar el código MIPS.